Introduction

We are a leading financial market and related data provider with a reputation for market-moving analysis. Our Data Science (DS) team, led by a former IBM Research scientist, consumes our own corporate data intake, sushi-style, and produces ad hoc and subscription-based machine learning analysis. The DSS team preprocesses lake and warehouse data independently but (a) other internal data factories are batch and stream processing dependent and (b) these largely untapped data reservoirs present latent shareholder value if they were integrated (joined) in analysis. In other words, moderately processed data answers questions that begin with "what" and "who"; competition is fierce in this arena. Data "integration" via joins opens a portal to higher margin questions "why" and "how" that stand on the shoulders of "when". Since data joins themselves are a process and all data eventually is unbounded, there is competitive pressure for processing timeliness achieved via greater use of stream processing. This paper proposes Change Data Capture (CDC) to promote stream processing of data joins that does not compromise architecture reliability, scale, and maintenance and which will yield a higher margin service mix and competitive advantage.

Basis for current proposal

For purposes of this discussion, we distribute generic low-margin data: securities prices and corporate events. Customers may also subscribe to higher-margin event analyses generated from those lower-margin data. Two examples illustrate: (1) corporate financial statements quantify aggregated corporate events: incremental production, sales and debt that yields balance sheet growth. These are generic data. Machine learning disintegrates these periodic event data to answer the why, the how, the when for (a) risk-adjusted earnings across various windows of time synched to periodic securities market price movement. (2) In theory, current financial market prices are aggregations of (a) prior price sequences and (b) the aforementioned corporate events that drive shareholder value and thus expected prices. Financial market price feeds are a generic data we distribute. Machine learning predicts theoretical price movement. As a periodic share price moves from 10 to 12, data science workflow looks back to bounded data haystacks for “why” needles, “how” needles and via periodic time stamps also pinpoints the approximate “when” needle in the hay stack. Periodic is the frequency since processing is mostly batch which aggregates separate data sets, (corporate event and market price e.g.), and joins them via time stamp to facilitate event analyses. Batch is the primary process and separate is the resting state of our data due to our architecture’s view of data as bounded sets. By altering our architecture, this CDC proposal shifts the “event” in “event analysis” closer to the data source, enabling our DS team to process generic events and prices first as unbounded immutable event logs. In Kleppmann’s analogy, “application state is what you get when you integrate an event stream over time”: our current architecture produces, and our DS team consumes “state(now)” on the extreme LHS of figure 11-6. Then, batch processes disintegrate this state and join it with other disintegrated sets via time stamp, and finally apply machine learning to form analytical value for customers. Our architecture operates on the RHS as well, distributing raw “stream(t)” at low margins to our customers.

$$state(now) = \int_{t=0}^{now} stream(t)\,\mathrm{d}t \qquad\qquad stream(t) = \frac{\mathrm{d}\,state(t)}{\mathrm{d}t}$$

Figure 11-6. The relationship between the current application state and an event stream.

Service offered – more timely analysis

By altering our architecture to incorporate CDC, we enable the DS team to perform joins at the spigot = “stream(t)” and “differentiate the state by time” cutting out the “integrand stream(t) dt” that creates bounded sets “state(now)”. This is good because it is those bounded sets that compel us to use batch processing to disintegrate before joining. Since our customers cannot move joins further upstream and work with logs in scale as we can, we present more timely analysis which few others can, and thus extract shareholder value for our organization.

Service offered – integrated analysis

The 1st derivative of state is on the RHS of figure 11-6 above. Kleppmann says the 2nd derivative of state has no meaning, but it is evident that with this architecture the partial derivative does have meaning and may be executed in scale for additional shareholder value. For the partial derivative, stream processing may relate “stream(t)” from two or more data streams and provide gradients for partial derivatives with respect to multiple drivers, not only time. For example, one stream(t) might be corporate revenues and another might be government jobs reports and a third might be weather or social media scrapings from the internet. This is what our head of data science is requesting when he asks for better "integrated" data.

Putting these 2 services together, timely integrated analsysis can be offered with scale as data is joined at the spigot and multiple data are joined in stream processing. Analysis continues ad hoc, but more frequently than batch process previously. Possibly as valuable, analysis may be appended to existing generic streams, as a form of meta data for greater high frequency data margins. Data flows can be engineered from logs to pipeline one ML into another ML for recommendation engines and those outputs appended to intraday feeds.

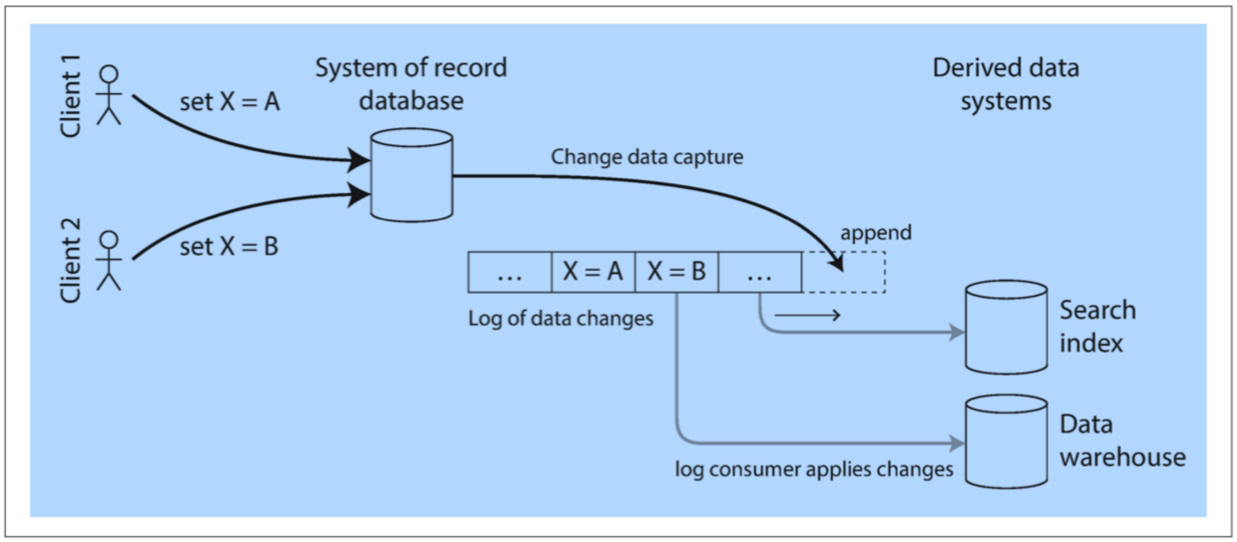

Figure 11-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

Architecture considerations - preliminary

The cost of incorporating change data logs may be borne by higher margins, but we must ensure that existing services are uninterrupted, and that architecture reliability and maintenance are unperturbed.

(1) Durability, legacy batch services, etc: In this proposal, the database viewed as a mutable state from immutable events is recorded in append only immutable logs. Thus, a replay of change data capture logs fosters replicated states and thus allow stream processing to continue producing derived data previously output from batch. Unlike AMQP and JMP message brokering, notification is asynchronous and non-destructive, enabling the log to be used as durable storage. With little rework, via the consumer offset, CDC enables snapshots for consistency and fault tolerant retries: one can quickly reconstruct state without replaying the whole log. Logs are kept in disk buffers and then archived. Archive compromises access, but consumer failure has days to pick up missing data given disk size. Compaction keeps only the latest primary key records but tombstones record deletion. Supported by Apache Kafka.

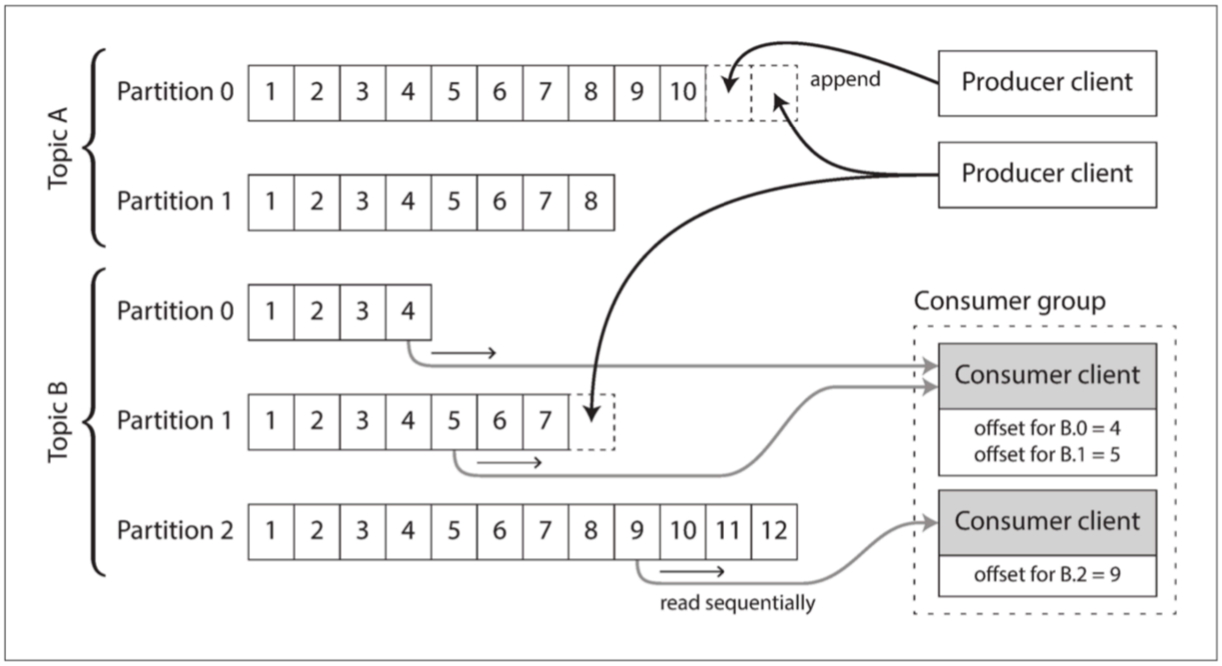

Figure 11-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

(2) Partitioning, replication, etc. For CDC, the producer appends to end of log and notifies the consumer which reads the log sequentially. Log is partitioned for parallel throughput from

different hosts and read independently by consumers. A topic is a group of partitions that carry messages of the same type. For the DS team, we cannot afford lost data, but we can afford more latency than securities transactions. We require some degree of correct time stamp for joins and subsequent machine learning, but do not need contemporaneous notification or transaction level timing. Ordering is required for partitioning and may be afforded by monotonically increasing offsets = sequence numbers but there are no ordering guarantees across global partitions; timestamps may help. The asynchronous model of brokered messages suits our needs: by employing logs, current messaging systems overhead that deploys backpressure and trades-off balanced throughput, latency, lost messages may be reduced. Apache Kafka is source technology.

(3) Leader follower. Consumer offset is similar to log sequence number of leader follower systems so changeover from that architecture will conceptually be addressable. Message broker serves as leader and consumer as follower. Immutable appendable logs must eventually be archived, compromising access for lagging consumer processes, but days of bounded-size are buffered. There are no side effects for other consuming processes: warehouses, indexes and other derived data are views on to system of record and so synch to DB are similar to the way replication logs are used with leader follower systems preserving order without race condition and less overhead. Scheme changes will need to be coded throughout to retain structure. Use case: Scale is found in LinkedIn's Databus and Facebook's Wormhole. PostgreSQL write-ahead log, MySQL binlog and MongoDB oplog facilitate change data log; Oracle too. Any latency we accept due to asynchronous notification might be compensated by placing replicas closer to consumers. In any case, our data is not time sensitive at that degree.

(5) Key issue: Consistency. Asynchronous communication brings inconsistency, e.g. reading own writes. Since we are distributing information not trading, eventual consistency suffices for internal and external customers. If read after write consistency were critically important, the log sequence number could be employed. As we compute from 2 data sets for integration, we may need to integrate into the same partition to ensure read is from the same "replica". Causally important states can be written to the same partition but not efficiently. This too might be important if we are viewing the causal nature of data that is out of time stamp confidence bounds. A lack of global ordering of writes is particularly a problem for distributed data. Dealing with these issues in application is not recommended for sustainability and maintainability due to complexity. This is a key issue as data is coming in from exchanges and corporations all over the world. If the ordering of events across streams is undetermined, the join for analysis and partitioning is non-deterministic. Input stream events may interleave in a different fashion when run twice. This is bad for retries. In warehouses, this slowly changing dimension is commonly addressed with unique identifiers, but again with data coming from all corners this may not be practical. Associating policy changes, government taxes or corporate leverage with price changes may become untenable were this a significant issue.

(6) Future considerations: Event sourcing. Since there is very little in the way of application logic generating useful data logic, event sourcing is not followed. We need to see the state of the system not the history of modifications. There is no useful information in the logic or why of data generated but there is value in synching the formation of data, to some degree. Were there actions taken based upon our data collected or distributed, then event sourcing might be a useful way for applications to interact with our improved data. If we could capture user

interaction with applications further upstream, e.g. in a trading application, causal analysis and event logs would be worth exploring.